Real-Time API

This API provides access to the **frame data** (location, class type and the tracking ID of each object within a frame), **occupancy changes** for each phase (virtual inductive loops that are called upon the presence of vehicles on the defined regions/loops), and **phase changes** at intersections (traffic light status). This API can be used to build more advanced applications on top of our traffic monitoring solution.

This API is using **gRPC** protocol for communication. You can either use our python client module to have access to the API or, build your own client based on the standards provided in this document.

Data structure

HyeprParameter

Our real-time server sends data as a **HyperParameter** object. **HyperParameter** is a proto message designed as follows:

```
message HyperParameter{
    string timestamp=1;
    Frame frame =2;
    PhaseChange phaseChange =3;
    OccupancyChange occupancyChange =4;
}
```

A HyperParameter can contain **frame**, **phase changes** and **occupancy changes** information. Also, it always includes a timestamp that is shared between all inner classes. The timestamp follows the **ISO 8601** standard format. HyperParameter helps to keep the order of simultaneous events. If we had used three separate streams for each of these inner events, we would have not been able to guarantee the simultaneous delivery of concurrent events. Each HyperParameter message can contain one to three of the events.

Frame

Each **Frame** message contains a *timestamp* and a list of road users' information detected by the sensor called **FrameObject**.

```
message Frame{
    string timestamp=1;
    repeated FrameObject objects = 2;

    // Only available on fussed data
    repeated LastTimeSeen lastTimesSeen =3;
}
```

A frame object contains the information about the detected objects at a specific timestamp in ISO 8601 fromat. Moreover, if data is received from a virtual sensor (an UDID that represents the fussed data between multiple sensors), it will contain **lastTimesSeen** attribute, which indicates the time of the last received frame from participanting sensors.

```
message LastTimeSeen{
    string udid=1;
    double time = 2;
}
```

Each **FrameObject** represents a road user detected in the frame. For each road user, we assign a unique id that is persistent in all the frames in which the object appears. We also send the location of the object in meter with respect to its distance to the sensor, width and length of the bounding box detected for the object, angle of the detected bounding box in radian, class type of the object, and some optional attributes like the speed of the objects, and the accuracy of the detection module. The optional values are not always guaranteed to be sent. Moreover, for virtual sensors that represents fusion between multiple edge boxes, **confidences** attribute will be shared to indicate the confidence level of detection in each participating sensors.

```
message FrameObject{
    string id=1;
    float centerX = 2;
    float centerY = 3;
    float width = 4;
    float length = 5;
    float rotation = 6;
    string classType=7;
    // Oprional
    float speed = 8;
    float centerZ =9;
    float height =10;
    float accuracy =11;
    // -----

    // Only available on fussed data
    repeated ConfidenceObeject confidences =12;

}

message ConfidenceObeject{
    float confidence =1;
    string udid =2;
}
```

Objects classtype can be one of the following cases:

Code	Class Types
0	Pedestrian (old version API)|
1	Vehicle (old version API)
2	Car
3	Van
4	Truck
5	Bus
6	SUV
7	Construction vehicle
8	Trailer
9	Tractor
10	Pedestrian
11	Person sitting
12	Cyclist
13	Bicycle
14	Person
15	Motorcycle
16	Rollerblader
17	Cyclist

PhaseChange

A **PhaseChange** event notifies clients of the traffic light changes in different phases. Each **PhaseChange** message contains a timestamp, a list of changed phases and a flag indicating whether this message includes information of all phases or not.

```
message PhaseChange{
    string timestamp  = 1;
    repeated Phase phases = 2;
    bool absolute  =3;
}
```

For the first connection to the server, a client receives a **PhaseChange** with a true *absolute* flag, including information of all the phases available at the intersection and their last status. This flag can be used for the initialization process of applications.

Moreover, each **PhaseChange** can include information on many phases that had a concurrent phase change. Each phase information is stored in the format of a **Phase** message. Each **Phase** message contains a phase number, the current status of the traffic light and the change timestamp (optional). If the **PhaseChange** *absolute* flag equals true, each of the phases inside of that message contains its own unique timestamp that indicates its latest change and its status. But if the flag equals false, the **Phase** message will not contain a timestamp.

```
message Phase {
    string phaseNumber=1;
```

```
    int32 status=2;
    string timestamp  = 3;
}
```

Phase status are, - 0 (*Invalid*) - 1 (*Green*) - 2 (*Yellow*) - 3 (*Red*)

OccupancyChange

Occupancy changes are sent as an **OccupancyChange** message. This message includes multiple occupancies information and an *absolute* flag similar to **PhaseChange**. Since the concurrent occurrence of occupancies is not high, **OccupancyChange** does not include a singular timestamp for all the occupancies inside. Instead, each **Occupancy** message contains a *timestamp* for itself. Occupancies are associated with phases. Therefore, to distinguish the occupancies, we use the label of the phase that the occupancy happened in. **Occupancy** messages have two *status* fields: true or false. A true value means the virtual loop is occupied, while a false value indicates that it is unoccupied.

```
message OccupancyChange{
    repeated Occupancy occupancies = 1;
    bool absolute =2;
}

message Occupancy{
    string phaseLabel=1;
    bool status=2;
    string timestamp  = 3;
}
```

Using the BCT python client module

In order to use our library, you need to install realtime_subscriber package using the provided wheel file. > python3 -m pip install bctRealtimeSubscriber-x.x.x-py3-none-any.whl

You need to import the ***BCTWSConnection*** class from either *Realtime_subscriber_api* or *Realtime_subscriber_service* depending on the shared repository. We have two types of client modules available. Based on your use case we share with you either the token-based module or the username and password-based module. The output data structure and overall process of both modules are similar. The only difference is the authentication process.

For testing purposes you might have been given an archive file that includes realtime data for a sepecfic range of time. In that case, please look at the Realtime_subscriber_archive sub-section.

Realtime_subscriber_api

```
from realtime_subscriber.Realtime_subscriber_api import BCTWSConnection

stream =
```

```
BCTWSConnection("<UDID>",username="<username>",password="<password>",singleto
n=False,subscriptions=
[BCTWSConnection.subscriptionOption.LOOP_CHANGE,BCTWSConnection.subscriptionO
ption.PHASE_CHANGE,BCTWSConnection.subscriptionOption.FRAME])
```

For creating an object from the *BCTWSConnection* class, you need to use the username and password shared with you and also the UDID of the sensor you intend to connect. The module communicates with our authentication service and fetches a JWT token that will be used in the background to communicate with our real-time servers. Moreover, you can indicate the subscription types of your connection. Based on your user level of access, you can fetch **Frame**, **PhaseChange**, **OccupancyChange** and/or **DetectionFrame** from the server.

Realtime_subscriber_service

```
from realtime_subscriber.Realtime_subscriber_service import BCTWSConnection
token = "<Token>"
stream =
BCTWSConnection("<UDID>",service="<service>",singleton=False,subscriptions=
[BCTWSConnection.subscriptionOption.FRAME],token=token)
```

If you are using the *BCTWSConnection* from *Realtime_subscriber_service*, you would need to provide the shared token and the service name associated with the token as the parameters of the object construction. Similar to *Realtime_subscriber_api* you can specify your desired subscription level.

Moreover, by assigning the singleton flag to True (default) or False you can change the format in which you receive the data. If the singleton is set to True, you would receive the data as a *hyperparameter* as soon as the server sends you a message.

```
while True:
    data = stream.get_hyperparameter()
    print (data.frame)
    print (data.occupancyChange)
    print (data.phaseChange)
```

Otherwise, if you set the flag to False, you would receive data separated for each event. You can use multithreading to process each event simultaneously.

```
    def getFrame():
        while True:
            data = stream.get_frame()
            print (data)
            print("\n\n")

    def getOccupancy():
        while True:
            data = stream.get_occupancy()
            print (data)
            print("\n\n")
```

```
    def getPhase():
        while True:
            data = stream.get_phase()
            print( data)
            print("\n\n")



    frame_thread =threading.Thread(target=getFrame)
    occupancy_thread= threading.Thread(target=getOccupancy)
    phase_thread = threading.Thread(target=getPhase)


    frame_thread.start()
    phase_thread.start()
    occupancy_thread.start()

    frame_thread.join()
    phase_thread.join()
    occupancy_thread .join()
```

Your connection to the server is a long-lived http2 connection and is persistent. You only need to create a connection object once and call *get_...()* functions to receive queued data on your end in the order of sent. This client module stores the received data from the server into FIFO queues in the background. You can specify the size of the queue in the construction of the module.

If you lose your connection to the server for any reason, right after reconnection, the server sends back all the data queued for your user. If your downtime exceeds the maximum length of the queue on the server, it first sends all the queued messages, then it sends a message containing *absolute* flags for both **OccupancyChange** events and **PhaseChange** events to notify you of the re-initiate state caused by the missing data.

Realtime_subscriber_archive

For testing purposes you might have been given an archive file that includes realtime data for a sepecfic range of time. In order to read the archive file you can use Realtime_subscriber_archive as follows:

```
from realtime_subscriber.Realtime_subscriber_archive import ArchiveReader

reader = ArchiveReader("<filename>.archive",fps=10)
while True:
    hyperParam = reader.get_hyperparameter()
    print (hyperParam.frame)
    print (hyperParam.occupancyChange)
    print (hyperParam.phaseChange)
```

Building a client module from scratch

This section describes our client module architecture and standard for those who wish to build a custom client module to communicate with our server.

Communication flow

The communication flow of our real-time service consists of three parts. 1. Authentication (If applicable) 2. Routing 3. Connecting

1. Authentication

(If you are eligible for connection to our servers as a service, you can skip this section)

The first step of the communication flow is generating a JWT token based on your credentials. You need to send your credentials to our authentication endpoint in order to receive a JWT access token and a refresh token. The received access token expires in 24 hours and the refresh token expires in 168 hours.

The endpoint to receive your access token is https://core.api.bluecity.ai/api/token/. You need to send a POST request to this endpoint with a username and password included in the body of your request. If your response status code is 200, you can expect output similar to the following structure.

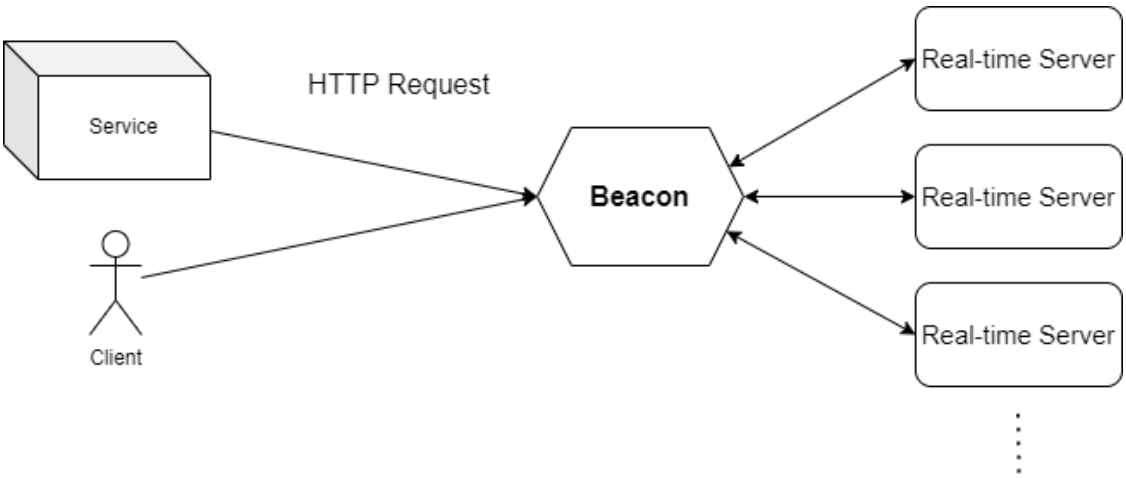
```
{
    "refresh": "<token>",
    "access": "<token>"
}
```

In order to refresh your access token, you can use the https://core.api.bluecity.ai/api/token/refresh/ endpoint. You would need to send your refresh token as a POST request in the body of your request.

Access to real-time servers is available through your access token.

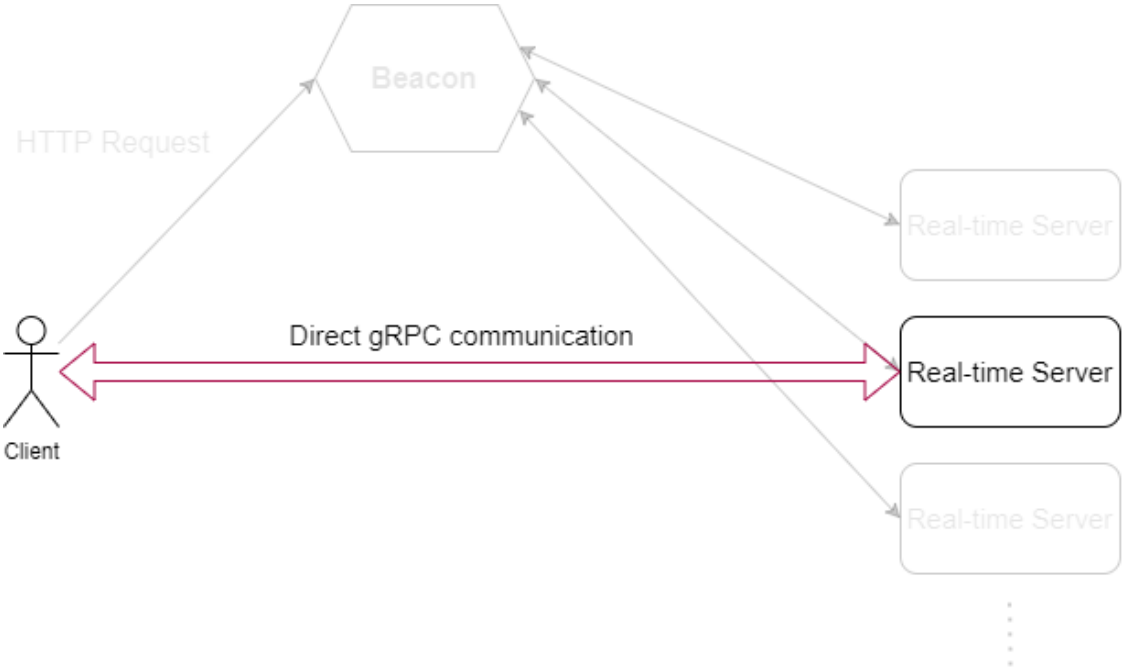
2. Routing

Real-time servers use a look-aside load balancing architecture. At the beginning of the connection, clients are required to ask **Beacon** (real-time server load balancer) for the correct real-time server endpoint and its SSL pub key for their desired sensor UDID.



Routing - Beacon

Clients can communicate directly with the assigned server after retrieving the endpoint and key.



Routing - Direct communication

In order to get the correct endpoint for your connection, you need to send an HTTP request (GET/POST) to one of our beacon addresses (https://realtime.us.beacon.1.api.bluecity.ai/route). Your request should contain the below information: - UDID: UDID of the desired sensor - token: Either your service access token or your JWT access token - type: This parameter indicates your connection type. 2 for access with JWT, 3 for access with service token

If you receive a 200 status code as your response, you can expect the following structure in the body of the response:

```
{
    "address":"<realtime server address>",
    "key":"<public key>"
}
```

Please be advised that if your connection with a real-time server goes down for any reason, you are expected to ask the **Beacon** again for an address and a key.

3. Connecting

Real-time servers communicate with clients using the gRPC protocol. If you decide to build your own client module, you can use the shared proto file to build the data structure in your desired language.

The communication with our server happens via **Subscriber** service.

```
service Subscriber{
    // this function should be called by clients.
    rpc subscribe(SubscriptionRequest) returns (stream HyperParameter) {}
    rpc ping(Empty) returns (Empty) {}
}
```

Subscirber service contains **subscribe** rpc. **subscribe** is a server streaming RPC where the client sends a **SubscriptionRequest** to the server and gets a stream of **HyperParameter**.

SubscriptionRequest contains an *initial* flag that indicates whether you are requesting for initial values or not.

```
message SubscriptionRequest{
    bool initial =1;
}
```

Connection to the real-time server needs to happen over a TSL channel. You can use the fetched address and pubkey to issue a secure connection to the server. Furthermore, in order to authenticate your connection with our real-time servers, you need to send your authentication credentials in the header/metadata of your request. A valid request to the server includes the following information:

- *bct-udid*: UDID of the desired sensor
- *token*: Authentication token (JWT or service access token)
- *type*: This parameter indicates your connection type. 2 for access with JWT, 3 for access with service token

If your credentials are valid you will have a long-lived http2 connection with our server.